Partial State in Dataflow-Based Materialized Views

Jon Gjengset — Doctoral Dissertation
jon@thesquareplanet.com / @jonhoo
My Committee

Robert Morris
(thesis advisor)

M. Frans Kaashoek

Sam Madden

Malte Schwarzkopf
Why are we here?

To make databases better.
Database 101

You take some tables.
Database 101

To **query**, do this:
Database 101

To update, do this:
Why are we here?

More orange work than blue.

But orange is often more common!
Why are we here?

Repeated, unnecessary orange work.
But Jon, caches.

Queries are now fast again!
Caches are great.

But caching is hard.
Automatic database caching.
Back to the title:

Partial State in Dataflow-Based Materialized Views
Back to the title:

Partial State in Dataflow-Based **Materialized Views**
Remembering Query Results

- Invented by the database community in the 1980s.
- Essentially “run the query and remember the result”.
- Key question is how to maintain the materialization:
  - What happens if the underlying data changes?
  - Should be incremental: don’t execute from scratch each time.
  - Maintain on write or on subsequent read?
Back to the title:

Partial State in **Dataflow-Based** Materialized Views
Push Changes to Views

- Dataflow has many definitions; here: data moves to compute.
  - Think “push-based computation”.
- Data changes propagate through graph of operators.
  - Here: relational operators like joins, aggregations, and filters.
- Each edge is a data dependency.
  - e.g., a join depends on its inputs.
- Messages are deltas:
  - Each delta is a full row with a positive (add) or negative (remove) sign.
CREATE MATERIALIZED VIEW StoryWithVC AS SELECT stories.*, COUNT(votes.user) AS votes FROM stories JOIN votes ON (votes.story_id = stories.id) GROUP BY stories.id;
CREATE MATERIALIZED VIEW StoryWithVC
AS SELECT stories.*,
    COUNT(votes.user) AS votes
FROM stories
JOIN votes
    ON (votes.story_id = stories.id)
GROUP BY stories.id;
CREATE MATERIALIZED VIEW StoryWithVC
AS SELECT stories.*, COUNT(votes.user) AS votes
FROM stories
JOIN votes
ON (votes.story_id = stories.id)
GROUP BY stories.id;
Example Dataflow Execution

CREATE MATERIALIZED VIEW 
    StoryWithVC
AS SELECT 
    stories.*, 
    COUNT(votes.user) AS votes 
FROM stories 
JOIN votes 
ON (votes.story_id = stories.id) 
GROUP BY stories.id;

SELECT * FROM StoryWithVC WHERE id = ?
CREATE MATERIALIZED VIEW
  StoryWithVC
AS SELECT
  stories.*,
  COUNT(votes.user) AS votes
FROM stories
JOIN votes
  ON (votes.story_id = stories.id)
GROUP BY stories.id;
CREATE MATERIALIZED VIEW StoryWithVC
AS SELECT stories.*, COUNT(votes.user) AS votes
FROM stories
JOIN votes
ON (votes.story_id = stories.id)
GROUP BY stories.id;
CREATE MATERIALIZED VIEW
  StoryWithVC
AS SELECT
  stories.*,
  COUNT(votes.user) AS votes
FROM stories
JOIN votes
  ON (votes.story_id = stories.id)
GROUP BY stories.id;
Example Dataflow Execution

CREATE MATERIALIZED VIEW StoryWithVC
AS SELECT stories.*, COUNT(votes.user) AS votes
FROM stories
JOIN votes
ON (votes.story_id = stories.id)
GROUP BY stories.id;
CREATE MATERIALIZED VIEW StoryWithVC
AS SELECT stories.*, COUNT(votes.user) AS votes
FROM stories
JOIN votes
ON (votes.story_id = stories.id)
GROUP BY stories.id;
Back to the title:

Partial State in Dataflow-Based Materialized Views
Learning to Forget

- Chances are that most entries in the view are not accessed.
  - Old and unpopular stories are wasting memory.
- Need to evict old entries, and only add new ones on demand.
- Three main contributions:
  - Notion of missing state in materialized views.
  - Upqueries to populate missing state using dataflow.
  - Implementation and evaluation of partial state in Noria.
CREATE MATERIALIZED VIEW StoryWithVC AS SELECT stories.*, COUNT(votes.user) AS votes FROM stories JOIN votes ON (votes.story_id = stories.id) GROUP BY stories.id;

SELECT * FROM StoryWithVC WHERE id = ?
CREATE MATERIALIZED VIEW StoryWithVC AS SELECT stories.*, COUNT(votes.user) AS votes FROM stories JOIN votes ON (votes.story_id = stories.id) GROUP BY stories.id WHERE stories.id = ?;
CREATE MATERIALIZED VIEW 
   StoryWithVC 
AS SELECT 
   stories.*, 
   COUNT(votes.user) AS votes 
FROM stories 
JOIN votes 
   ON (votes.story_id = stories.id) 
GROUP BY stories.id 
WHERE stories.id = 7;
Misses Trigger Upqueries

CREATE MATERIALIZED VIEW StoryWithVC AS SELECT stories.*, COUNT(votes.user) AS votes FROM stories JOIN votes ON (votes.story_id = stories.id) GROUP BY stories.id WHERE stories.id = 7;
 CREATE MATERIALIZED VIEW StoryWithVC AS SELECT stories.*, COUNT(votes.user) AS votes FROM stories JOIN votes ON (votes.story_id = stories.id) GROUP BY stories.id WHERE stories.id = 7;
CREATE MATERIALIZED VIEW StoryWithVC
AS SELECT stories.*, COUNT(votes.user) AS votes
FROM stories
JOIN votes
ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = 7;
CREATE MATERIALIZED VIEW StoryWithVC AS
SELECT stories.*,
    COUNT(votes.user) AS votes
FROM stories
JOIN votes
    ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = 7;
CREATE MATERIALIZED VIEW StoryWithVC AS SELECT stories.*, COUNT(votes.user) AS votes FROM stories JOIN votes ON (votes.story_id = stories.id) GROUP BY stories.id WHERE stories.id = 7;
CREATE MATERIALIZED VIEW StoryWithVC
AS SELECT stories.*, COUNT(votes.user) AS votes
FROM stories
JOIN votes
ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = 7;
CREATE MATERIALIZED VIEW StoryWithVC AS
SELECT stories.*,
COUNT(votes.user) AS votes
FROM stories
JOIN votes
ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = 7;
CREATE MATERIALIZED VIEW StoryWithVC AS
  SELECT stories.*,
  COUNT(votes.user) AS votes
FROM stories
JOIN votes
  ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = 7;
CREATE MATERIALIZED VIEW StoryWithVC AS SELECT stories.*, COUNT(votes.user) AS votes FROM stories JOIN votes ON (votes.story_id = stories.id) GROUP BY stories.id WHERE stories.id = ?;
CREATE MATERIALIZED VIEW StoryWithVC AS SELECT stories.*, COUNT(votes.user) AS votes FROM stories JOIN votes ON (votes.story_id = stories.id) GROUP BY stories.id WHERE stories.id = ?;
CREATE MATERIALIZED VIEW StoryWithVC
AS SELECT stories.*, COUNT(votes.user) AS votes
FROM stories
JOIN votes
ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = ?;
Intermission

Related work
Materialized View Maintenance

- Primarily targets analytics workloads ➔ infrequent reads.
- Little or no support for on-demand queries.
- No support for eviction.
Automated Caching Systems

- Few are general-purpose.
- Many only support invalidation, not updates.
- Often limited to specific database interaction, not general SQL.
Dataflow and Stream Processing

- Usually focused on write performance.
- Focus on strong consistency at the cost of read latency.
- Limited support for on-demand compute & eviction.
Are we done?
In Practice, Things are Hard

- Must ensure that data changes take effect exactly once.
- Traditionally easy, but hard in this model because:
  - Upqueries hold past state which may be concurrently updated.
  - Updates may be discarded early.
- Many hazards (see thesis), but we’ll focus on one.
Incongruent Join Evictions
What is an Incongruent Join?

CREATE MATERIALIZED VIEW StoriesWithAuthor AS SELECT stories.*, users.name AS aname, FROM stories JOIN users ON (stories.author = users.id) WHERE stories.id = ?;
CREATE MATERIALIZED VIEW StoriesWithAuthor AS SELECT stories.*, users.name AS aname, FROM stories JOIN users ON (stories.author = users.id) WHERE stories.id = ?;
CREATE MATERIALIZED VIEW StoriesWithAuthor AS SELECT stories.*, users.name AS aname, FROM stories JOIN users ON (stories.author = users.id) WHERE stories.id = 7;
CREATE MATERIALIZED VIEW StoriesWithAuthor AS SELECT stories.*, users.name AS aname, FROM stories JOIN users ON (stories.author = users.id) WHERE stories.id = 7;
CREATE MATERIALIZED VIEW StoriesWithAuthor AS SELECT stories.*, users.name AS aname, FROM stories JOIN users ON (stories.author = users.id) WHERE stories.id = 7;
Upquery Works Correctly

CREATE MATERIALIZED VIEW StoriesWithAuthor
AS SELECT
    stories.*,
    users.name AS aname,
FROM stories
JOIN users
    ON (stories.author = users.id)
WHERE stories.id = 7;

7, stories.*, aname: Lena
Recall This Figure?
CREATE MATERIALIZED VIEW StoriesWithAuthor AS SELECT stories.*, users.name AS aname, FROM stories JOIN users ON (stories.author = users.id) WHERE stories.id = 7;

- 7, ..., author: 42
+ 7, ..., author: 43
CREATE MATERIALIZED VIEW StoriesWithAuthor AS SELECT stories.*, users.name AS aname, FROM stories JOIN users ON (stories.author = users.id) WHERE stories.id = 7;
CREATE MATERIALIZED VIEW StoriesWithAuthor AS SELECT stories.*, users.name AS aname, FROM stories JOIN users ON (stories.author = users.id) WHERE stories.id = 7;
CREATE MATERIALIZED VIEW StoriesWithAuthor AS SELECT stories.*, users.name AS aname, FROM stories JOIN users ON (stories.author = users.id) WHERE stories.id = 7;
CREATE MATERIALIZED VIEW StoriesWithAuthor AS
SELECT stories.*,
    users.name AS aname,
FROM stories
JOIN users
ON (stories.author = users.id)
WHERE stories.id = 7;
What Do We Do?

- Cannot produce needed update!
- Cannot forward just the negative.
- Cannot drop update altogether.
What Do We Do?

- Cannot produce needed update!
- Cannot forward just the negative.
- Cannot drop update altogether.
- Fill missing state?
What Do We Do?

- Cannot produce needed update!
- Cannot forward just the negative.
- Cannot drop update altogether.

---

Fill missing state?

- 7, ..., aname: Lena
  + 7, ..., author: 43

43, ???
What Do We Do?

- Cannot produce needed update!
- Cannot forward just the negative.
- Cannot drop update altogether.
  Fill missing state?
- **Evict** downstream state.
What Do We Do?

- Cannot produce needed update!
- Cannot forward just the negative.
- Cannot drop update altogether.
  - Fill missing state?
  - **Evict** downstream state.
- Next query fills it again.
Does it work?
Need a Realistic Test Subject

- Lobste.rs: a Hacker News-like news aggregator.
  - Users submit stories, vote for and comment on them, etc.
  - Open-source, so we can see the queries.
  - Data statistics available, so we know the workload.
- Workload generator: synthesize Lobste.rs-like requests.
Throughput

Fixed available resources.
Throughput

Fixed available resources.
Throughput

Fixed available resources.
Memory use

Fixed throughput & runtime.
Noria vs. cache
vs. Redis

Idealized cache workload.
vs. Redis

Idealized cache workload.

*Redis is single-threaded, so 16x is extrapolated.*
vs. Redis

Idealized cache workload.

Redis is single-threaded, so 16x is extrapolated.
Wrapping things up
Future work

Noria is neither perfect nor complete.

- Range queries, cursors, time-windowed operators.
- Upstream database integration.
- Maintaining downstream views.
- Fault tolerance.
Acknowledgements
Parallel & Distributed Operating Systems Group
Conclusion

My thesis enables **materialized views** to be used as **caches**.

It does so by allowing state to be **missing** from materializations, and using **upqueries** to populate missing state on demand.

The resulting system provides **automated** caching for SQL queries, and reduces the need for complex, ad hoc caching logic.

Thank you — please ask questions!
jon@thesquareplanet.com
Backup slides
<table>
<thead>
<tr>
<th>Page</th>
<th>%</th>
<th>W</th>
<th>Q</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Story</td>
<td>55.8</td>
<td>1</td>
<td>14</td>
<td>Renders an individual story’s page, including its popularity score, comments,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>and the scores of its comments.</td>
</tr>
<tr>
<td>Front page</td>
<td>30.1</td>
<td>0</td>
<td>14</td>
<td>Lists the 25 most highly scored stories, along with their authors and scores.</td>
</tr>
<tr>
<td>User</td>
<td>6.7</td>
<td>0</td>
<td>7</td>
<td>Renders a user summary page, including what story “tags” they contribute to.</td>
</tr>
<tr>
<td>Comments</td>
<td>4.7</td>
<td>0</td>
<td>9</td>
<td>Like the front page, but for comments.</td>
</tr>
<tr>
<td>Recent</td>
<td>1.0</td>
<td>0</td>
<td>14</td>
<td>25 most recently added stories, along with their authors and scores.</td>
</tr>
<tr>
<td>Vote</td>
<td>1.2</td>
<td>1</td>
<td>2</td>
<td>Vote up/down a given comment or story.</td>
</tr>
<tr>
<td>Comment</td>
<td>0.4</td>
<td>2</td>
<td>5</td>
<td>Add a new comment to a story.</td>
</tr>
</tbody>
</table>

Table 6.1.: Pages in Lobsters. % indicates the percentage of requests that load the given page. W is the number of writes performed by a given page. Q is the number of (read) queries a page issues.